CSE 168: Rendering Algorithms

Steve Rotenberg

UCSD

Spring 2017

CSE168

- Rendering Algorithms
- Instructor: Steve Rotenberg (srotenberg@ucsd.edu)
- TA: Matteo Mannino (mtmannin@eng.ucsd.edu)
- Lecture: WLH 2205 (MW 5:00 – 6:20pm)
- Office: EBU3 4106 (MW 3:50 – 4:50pm)
- Lab: EBU3 basement
- Discussion: York 4080A (M 11:00 – 11:50am)
- Web page:
  - https://cseweb.ucsd.edu/classes/sp17/cse168-a/index.html

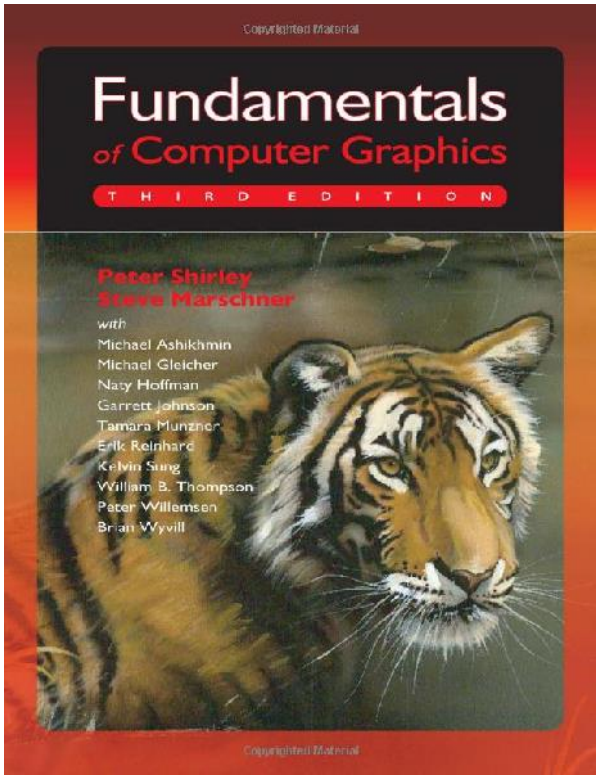
Prerequisites

- CSE167 or equivalent introduction to 3D graphics
- Familiarity with:
  - Vectors (dot products, cross products, etc.)
  - Matrices (4x4 homogeneous transforms, etc.)
  - Polygon rendering
  - Basic lighting (normals, Phong, etc.)
  - Object oriented programming

Reading

- Fundamentals of Computer Graphics, Peter Shirley, Steve Marschner
- 2nd or 3rd edition

- The book is optional for this class. It covers many of the topics we will go over in class, but we won't follow the book exactly.



Programming Projects

- Project 1: Due 4/12 (Wednesday, week 2)
  - Basic ray tracer: render a box with basic lighting
- Project 2: Due 4/26 (Wednesday, week 4)
  - Add spatial data structure to efficiently render complex geometry with shadows
- Project 3: Due 5/10 (Wednesday, week 6)
  - Add reflections & materials
- Project 4: Due 5/24 (Wednesday, week 8)
  - Add path tracing
- Project 5: Due 6/16 (Friday, finals week)
  - Add your own choice of features and render a final image

Programming Projects

- You can use any programming language & operating system that you choose
- However, I would recommend using C++ for the following reasons:
  - Rendering is slow and you want a compiled language that will give you the chance of the best performance
  - Graphics tends to fit very naturally into an object oriented framework
  - There are several situations where you will need to make use of virtual functions and derived classes

Programming Assignment Turn-In

- The project must be shown to the instructor or TA before 5:00pm on the due date (when class starts)
- They will both be in the lab from 2:00-4:50 on due days, but you can turn them in early as well
- If necessary, projects can be turned in immediately after class on due days if you speak to the instructor before hand (for example if there are too many projects to grade in time)
- If you finish on time but for some strange reason can't turn it in personally, you can email the code and images to the instructor *and* TA and demo it personally some time in the following week for full credit
- If you don't finish on time, you can turn what you have in for partial credit. Either way, you can turn it in late during the following week for -4 points. So for example on a 15 point assignment, you can turn it in for partial credit and get 6, but then finish it and turn it in late for up to 11 points
- Anything after 1 week can still be turned in but for -8 points
- Note that all projects build upon each other so you have to do them eventually…

Final Project

- For the final project, you can implement some rendering features of your choice and render out a final image
- You will have to do a 5 minute presentation to the class during the 3 hour ‘final’

Grading

- Project 1: 15%
- Project 2: 15%
- Project 3: 15%
- Project 4: 15%
- Project 5: 15%
- Midterm: 10%
- Final: 15%

Course Outline

1. Introduction
2. Ray Intersections & Scenes
3. Fresnel Surfaces
4. Materials
5. Shadows & Area Lights
6. Spatial Data Structures
7. Antialiasing
8. Texture mapping
9. Random sampling
10. (Midterm)
11. BRDFs
12. Cameras
13. Adaptive sampling
14. Light physics
15. Path tracing
16. Volumetric rendering
17. High dynamic range (HDR)
18. Bezier surfaces & tessellation
19. Procedural texturing
20. (In-class Final)

Rendering Overview

Computer Graphics

- The subject of computer graphics has grown over its 50 year history to include a wide range of topics
- Still, however, it is often convenient to divide it into three traditional sub-topics: *modeling*, *rendering*, and *animation*

Modeling

- Modeling deals with geometry representation, creation, and analysis
- The subject of modeling includes:
  - Techniques used to specify geometry (triangles, curved surfaces, implicit surfaces, point clouds…)
  - Operations used to generate geometry (extrusions, tessellations, Boolean, L-systems…)
  - Higher level procedural modeling (procedural plants, terrain generation…)
  - Techniques for acquiring models from the real world (laser scanning, photographic techniques…)

Animation

- Animation is the subject of movement and change over time.
- Some topics include:
  - Matrix & quaternion manipulation
  - Character animation (skeletons, skinning, blending, state machines…)
  - Physics simulation (particles, rigid bodies, fluid dynamics, deformable bodies, fracture…)
  - Dynamic visual effects, etc.

Rendering

- Rendering is the process of generating a 2D image from 3D geometry, lights, materials, and camera information
- Rendering could be further split into sub-topics: *photoreal rendering*, and *non-photoreal rendering (NPR)*
- Or another way to divide it up would be into *realtime rendering*, and *non-realtime rendering*
- For this discussion, I'll break it into three topics: *photoreal*, *NPR*, and *realtime*

Rendering

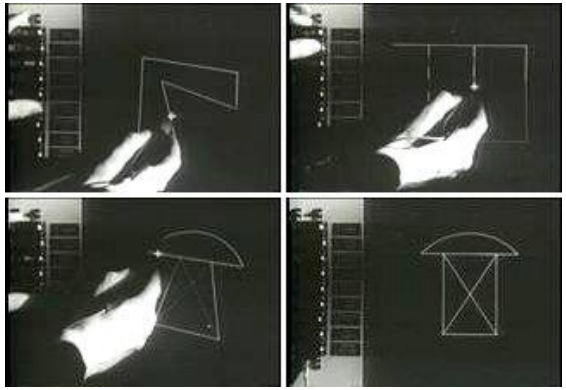
- Realtime:
  - The goal of realtime rendering is to generate a high quality image as quickly as possible (typically around 1/60th of a second)
  - This is generally based on special-purpose hardware (GPUs) and makes use of shader programming and other graphics-specific languages (OpenGL, Direct3D, GLSL, Cg…)
  - Inspired by physics and photorealism, but compromises are required to run fast
  - Typically used for video games and other interactive applications
- Photoreal:
  - Photoreal rendering refers to the goal of making an image that is indistinguishable from a photograph
  - Photoreal rendering is based on a simulation of the actual physics of light
  - *General-purpose* photoreal rendering was a major goal of the graphics research community and *practical* solutions didn't really exist until around the year 2000
- Non-photoreal:
  - NPR refers to all other types of rendering where the goal is not to achieve photorealism
  - This mainly includes various artistic types of rendering such as methods that imitate the appearance artistic media like pencils, watercolors, and oil paints, or artistic styles such as impressionism, or even methods that imitate the styles of specific artists
  - NPR also includes methods for graphical display for non-artistic purposes, such as clear illustration of complex mechanical designs, etc.

CSE168: Rendering Algorithms

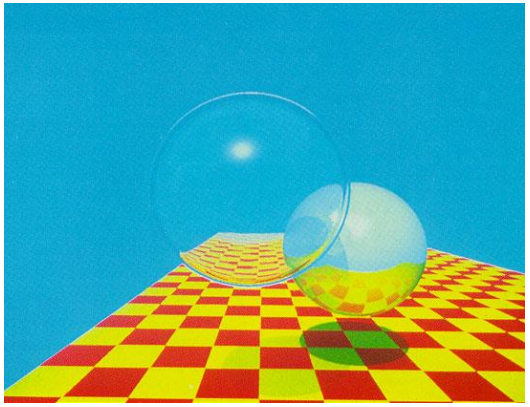
- In this class, we will focus mainly on photoreal rendering algorithms
- We may have a little time to spend on NPR, but we will not be spending any time on realtime rendering
- There have been many approaches to photoreal rendering developed in the past, however, one particular class of algorithms has proven to be the most successful and general-purpose
- These are the *ray-based* approaches that evolved from the original *ray tracing* algorithm, such as *path tracing* and *photon mapping* and these will be the main focus of the class

History

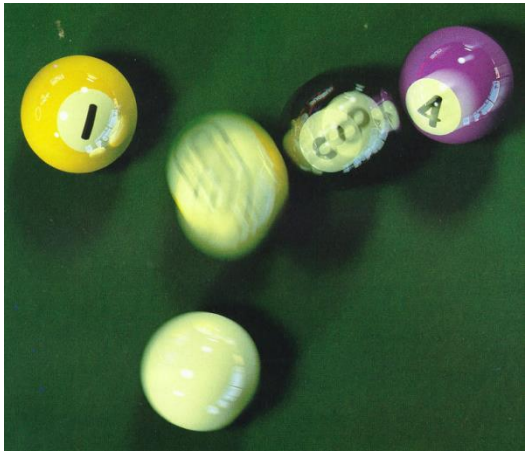
- This is a timeline of some key developments in ray-based photoreal rendering:



- 1963: ‘Sketchpad’, a project at MIT led by Ivan Sutherland, that is often considered the birth of computer graphics
- 1980: ‘Ray-Tracing’, a rendering algorithm developed by Turner Whitted that greatly improved the quality of rendered images by adding complex shadows, reflections, and refraction

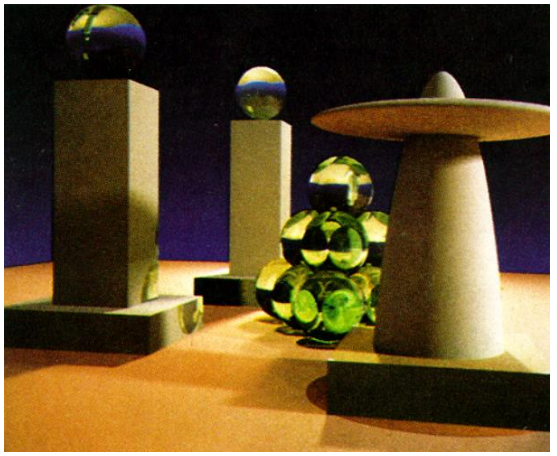


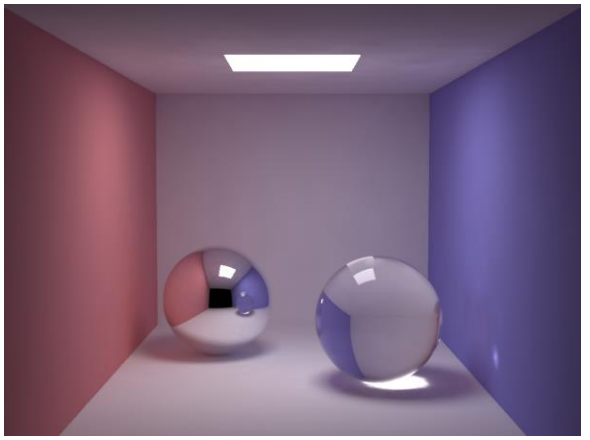
- 1984: ‘Distribution Tracing’, an extension to ray tracing that traced a distribution of multiple rays to allow for soft and blurry effects such as soft shadows, blurry reflections, motion blur, and camera focus

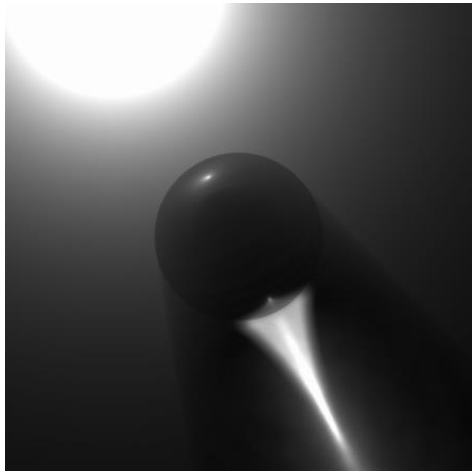


History

- 1986: ‘Path Tracing’, a method introduced in James Kajiya’s seminal paper called ‘The Rendering Equation’, that further extended ray tracing to become the first truly photoreal rendering algorithm, capable of all previous effects plus full global illumination and diffuse light bouncing. Path tracing was capable of great things, but was very slow on the hardware of the day
- 1995: ‘Photon Mapping’, a method introduced by UCSD’s professor Henrik Wann Jensen that works in conjunction with all previous ray based methods. Photon mapping improves the performance of many rendering situations and also makes it much more practical to handle focused light effects such as ‘caustics’ created by curved glass or mirrors, or the dancing lights at the bottom of a swimming pool
- 1998: Volumetric Photon Mapping: The photon mapping algorithm was extended to handle volumetric scattering in media like fog



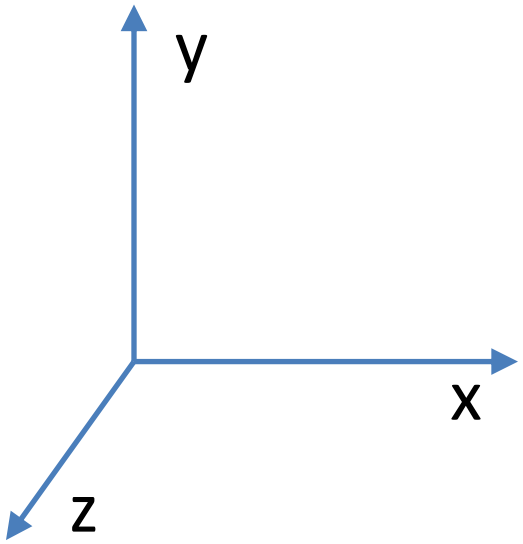




Linear Algebra Review

Coordinate Systems

- Right handed coordinate system



Vector Arithmetic

$$\mathbf{a} = [a_x \quad a_y \quad a_z]$$

$$\mathbf{b} = [b_x \quad b_y \quad b_z]$$

$$\mathbf{a} + \mathbf{b} = [a_x + b_x \quad a_y + b_y \quad a_z + b_z]$$

$$\mathbf{a} - \mathbf{b} = [a_x - b_x \quad a_y - b_y \quad a_z - b_z]$$

$$-\mathbf{a} = [-a_x \quad -a_y \quad -a_z]$$

$$s\mathbf{a} = [sa_x \quad sa_y \quad sa_z]$$

Vector Magnitude

- The magnitude (length) of a vector is:

$$|\mathbf{v}| = \sqrt{{v_x}^2 + {v_y}^2 + {v_z}^2}$$

- A vector with length=1.0 is called a *unit vector*
- We can also *normalize* a vector to make it a unit vector:

$$\frac{\mathbf{v}}{|\mathbf{v}|}$$

Dot Product

$$\mathbf{a} \cdot \mathbf{b} = \sum a_i b_i$$

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z$$

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}|\cos\theta$$

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b}$$

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix}$$

Properties of the Dot Product

- $\mathbf{a} \cdot \mathbf{b}$ is a scalar value that tells us about the relationship of two vectors
- If the dot product is positive, it means the angle between the vectors is less than 90 degrees and if its negative, the angle is more than 90 degrees
- If the dot product is 0, then the two vectors are either perpendicular or one or both are degenerate (0,0,0)

Cross Product

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} i & j & k \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix}$$

$$\mathbf{a} \times \mathbf{b} = [a_y b_z - a_z b_y \quad a_z b_x - a_x b_z \quad a_x b_y - a_y b_x]$$

Properties of the Cross Product

- $\mathbf{a} \times \mathbf{b}$ is a *vector* perpendicular to both **a** and **b**, in the direction defined by the right hand rule

- The *magnitude* of the cross product:

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}|\sin\theta$$

$|\mathbf{a} \times \mathbf{b}|$ = area of parallelogram **ab**

$|\mathbf{a} \times \mathbf{b}| = 0$ if **a** and **b** are parallel

Translation

- Let's say we have a 3D model that has an array of position vectors describing its shape: $\mathbf{v}_n$ where $0 \leq n < \text{NumVerts}$
- Say we want to move our 3D model from its current location to somewhere else…
- We want to compute a new array of positions $\mathbf{v'}_n$ representing the new location
- If the vector **d** represents the relative offset that we want to move our object by, then we can compute $\mathbf{v'}_n = \mathbf{v}_n + \mathbf{d}$ to compute the new array of positions

Transformations

$$\mathbf{v}'_n = \mathbf{v}_n + \mathbf{d}$$

- This translation represents a very simple example of an object *transformation*
- The result is that the entire object gets moved or *translated* by **d**
- From now on, we will drop the $_n$ subscript and just write

$$\mathbf{v}' = \mathbf{v} + \mathbf{d}$$

- Just keep in mind that this is actually a loop over several *different* $\mathbf{v}_n$ vectors applying the *same* vector **d** every time

Rotation

- Now, let's rotate the object in the *xy* plane by an angle θ, as if we were spinning it around the *z* axis

$$v'_x = v_x \cos\theta - v_y \sin\theta$$
$$v'_y = v_x \sin\theta + v_y \cos\theta$$
$$v'_z = v_z$$

- Note: a *positive* rotation will rotate the object *counterclockwise* when the rotation axis (z) is pointing *towards* the observer

Rotation

$$v'_x = v_x \cos\theta - v_y \sin\theta$$
$$v'_y = v_x \sin\theta + v_y \cos\theta$$
$$v'_z = v_z$$

- We can re-write this in matrix form as:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix}$$

- Or just:

$$\mathbf{v}' = \mathbf{M} \cdot \mathbf{v}$$

Matrix Transformations

- We can rotate a vector by multiplying it by a 3x3 rotation matrix
- We can also do other *linear transformations* with a 3x3 matrix such as:
  - Uniform & non-uniform scale
  - Shear (i.e., turning a rectangle into a parallelogram)
  - Reflection
- However, we can't use a 3x3 matrix to perform a translation

Homogeneous Transforms

- So, in computer graphics, we use 4x4 *homogeneous matrices* to combine translations with rotations (and shears, scales, and reflections)
- We can also integrate viewing transformations such as perspective projections into this system, however it turns out that we won't need to do that in this class, and all of our 4x4 transformation matrices will have the form:

$$\mathbf{M} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Positions & Directions

- Both positions and directions can be represented as 3D vectors
- When we transform a 3D vector with a 4x4 matrix, we have to turn the 3D vector into a 4D vector
- For positions, we put a 1 in the 4^{th} coordinate (w-coordinate), and for directions, we put a 0 in the 4^{th} coordinate
- The result is that a position gets rotated & translated by the matrix, but a direction only gets rotated

Positions & Directions

$$\mathbf{p} = \begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix} \rightarrow \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix} \qquad \mathbf{p}' = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} p_x \\ p_y \\ p_z \\ 1 \end{bmatrix}$$

$$\mathbf{n} = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix} \rightarrow \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix} \qquad \mathbf{n}' = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} n_x \\ n_y \\ n_z \\ 0 \end{bmatrix}$$

Object Space

- The space that an object is defined in is called *object space* or *local space*
- Usually, the object is located at or near the origin and is aligned with the xyz axes in some reasonable way
- The units in this space can be whatever we choose (i.e., meters, etc.)
- A 3D object would be stored on disk and in memory in this coordinate system
- When we go to draw the object, we will want to transform it into a different space

World Space

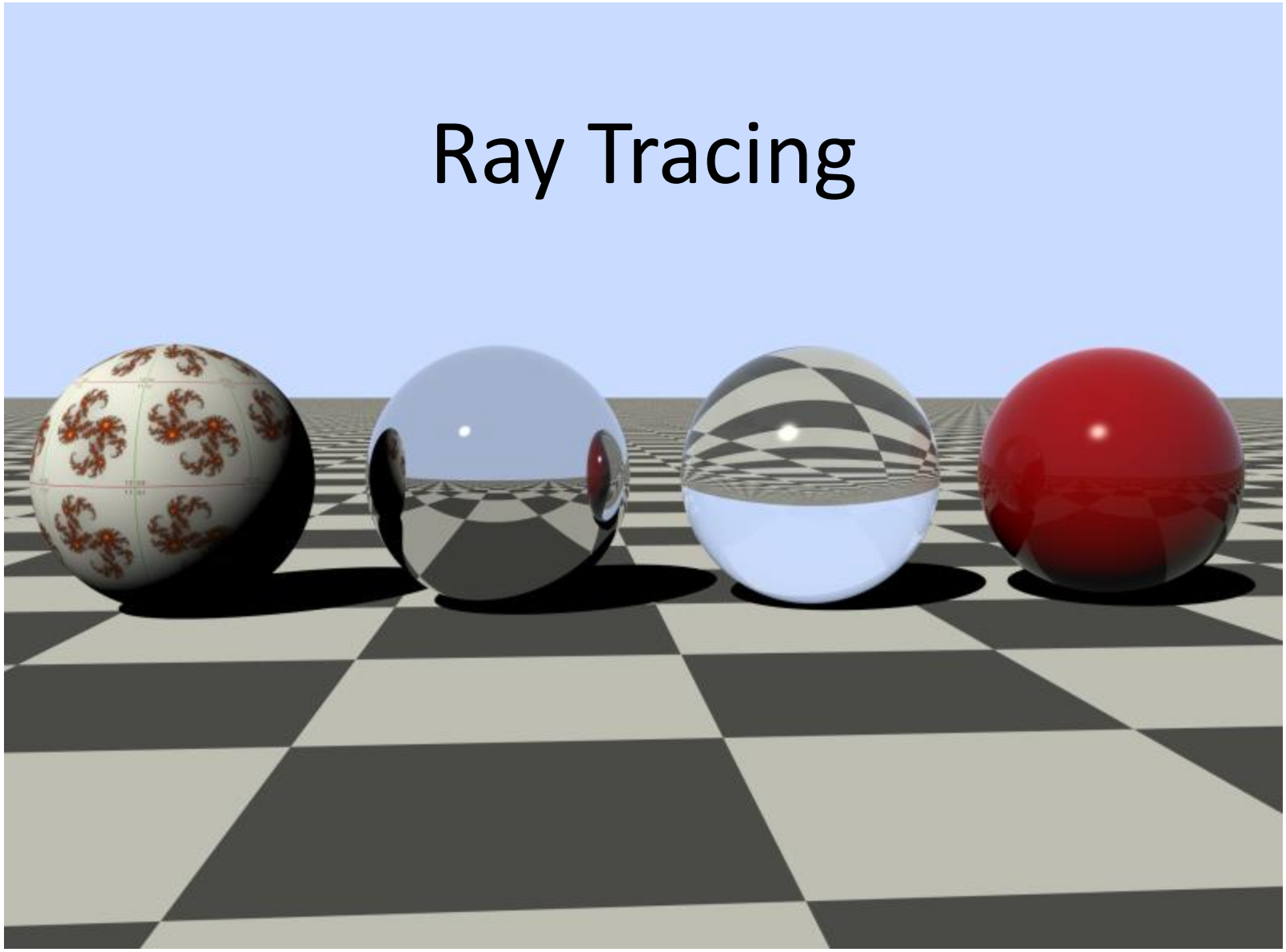
- We will define a new space called *world space* or *global space*
- This space represents a 3D world or scene and may contain several objects placed in various locations
- Every object in the world needs a matrix that transforms its vertices from its own object space into this world space
- We will call this the object's *world matrix*, or often, we will just call it the object's matrix
- For example, if we have 100 chairs in the room, we only need to store the object space data for the chair once, and we can use 100 different matrices to transform the chair model into 100 locations in the world

ABCD Vectors

$$\mathbf{M} = \begin{bmatrix} a_x & b_x & c_x & d_x \\ a_y & b_y & c_y & d_y \\ a_z & b_z & c_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- As the bottom row of our 4x4 matrices will always be 0 0 0 1, we can really think of the matrix as having only 12 relevant numbers, broken into 4 vectors, **a**, **b**, **c**, and **d**
- Vector **d** represents the translation of the matrix
- If we think of the matrix as transforming from object space to world space, then the **a** vector represents the object's x-axis rotated into world space, **b** is its y-axis in world space and **c** is its z-axis

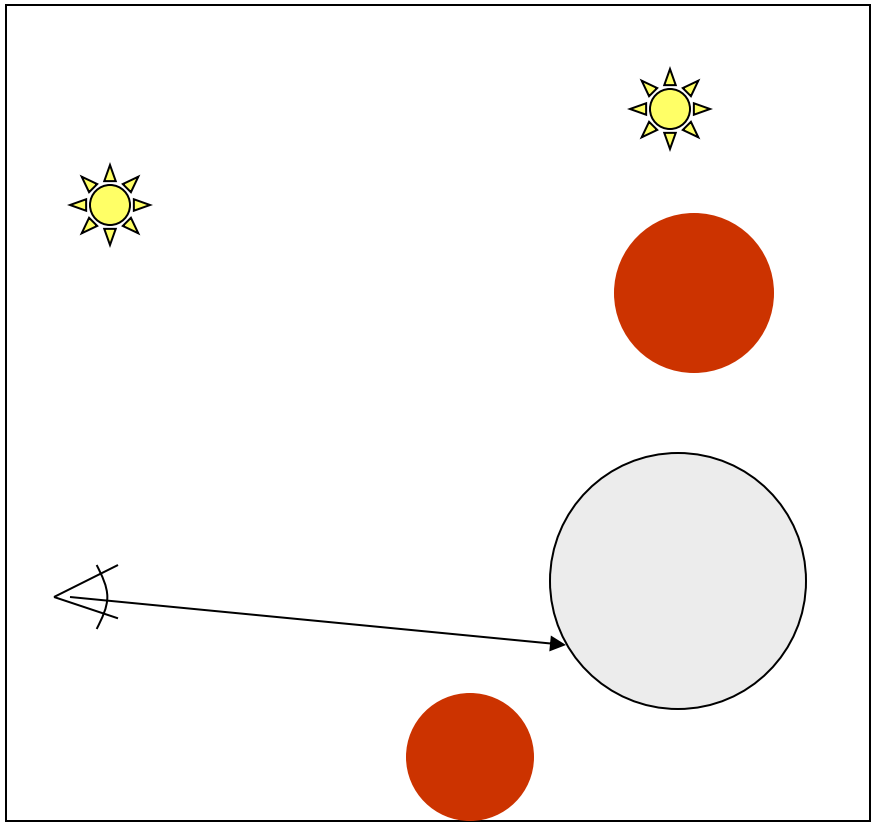
Ray Tracing

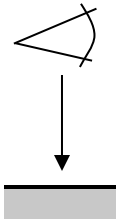


Ray Tracing

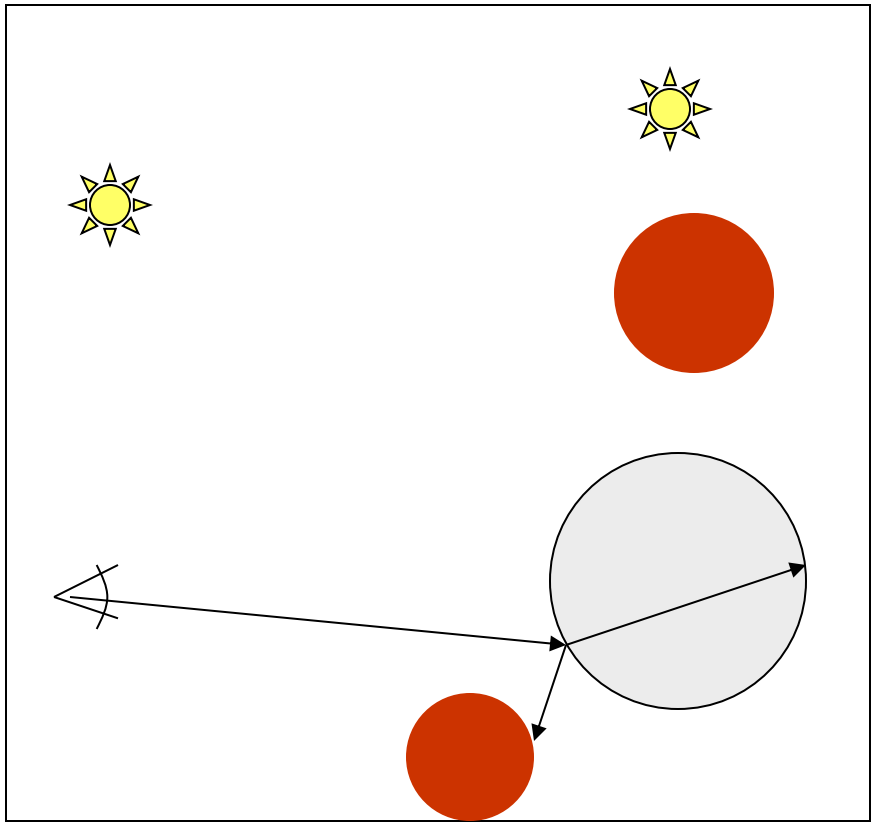
- Ray tracing involves tracing virtual rays throughout a 3D environment to determine object visibility and lighting
- Classic ray tracing shoots rays out from the camera position through each pixel to determine the first surface (triangle) hit
- From there, other rays can be spawned from the intersection point towards each light source to determine if the point is lit or in shadow
- Additional rays can be traced to simulate reflections off of mirrored surfaces or refraction through surfaces such as glass

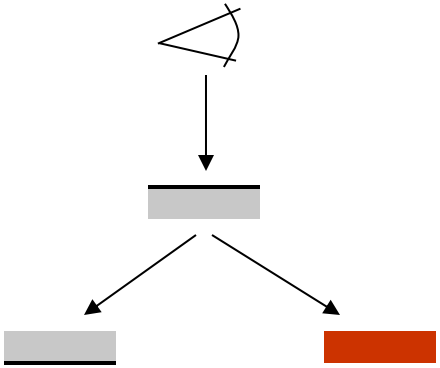
Recursive Ray Tracing



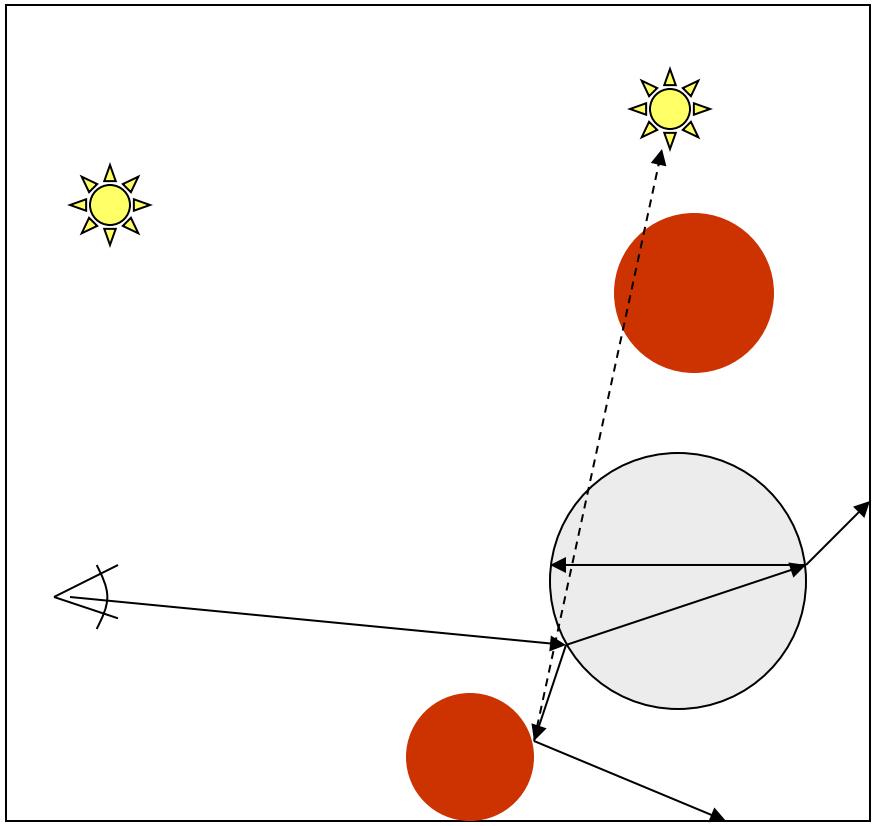


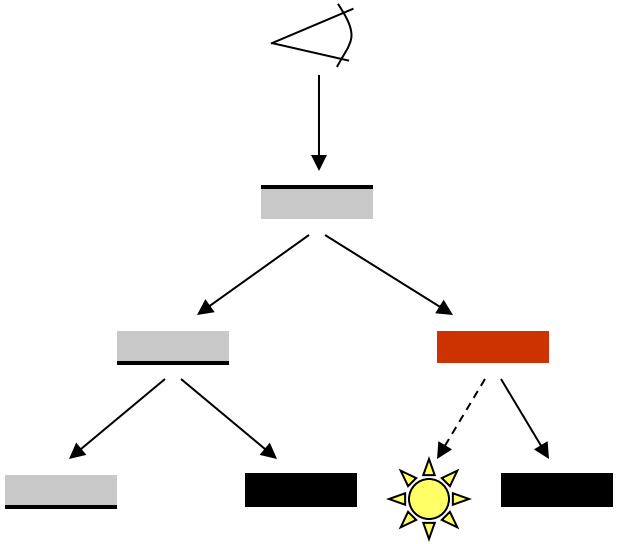
Recursive Ray Tracing



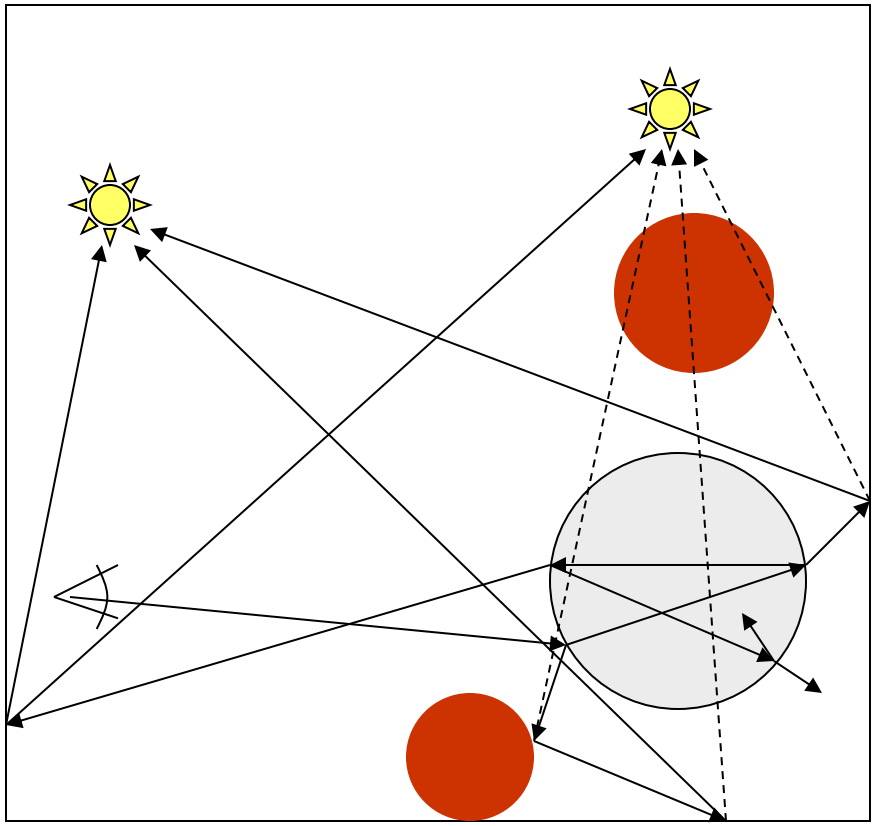


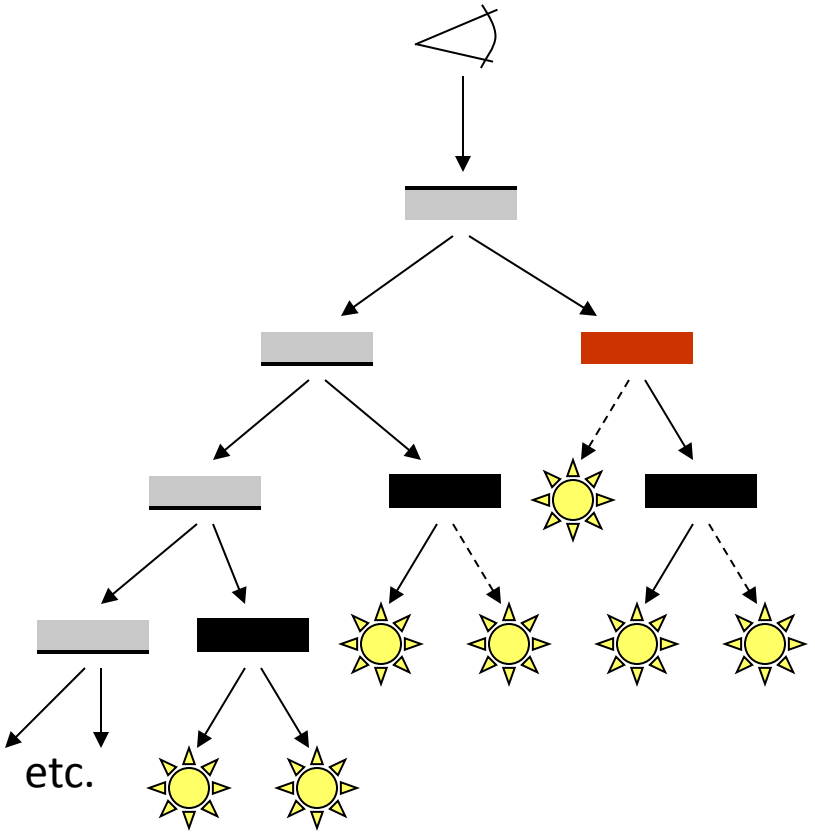
Recursive Ray Tracing





Recursive Ray Tracing





Rays

- We think of a ray as starting at an origin **p** and then shooting off infinitely in some direction **d**

$$\mathbf{r}(t) = \mathbf{p} + t\mathbf{d}$$

- Where **r**(*t*) is a function representing the ray and *t* is the distance traveled along the ray
- Usually, when we're rendering, we're interested in finding the first surface hit along the ray's travel from the origin, or the intersection with the smallest value of *t* (but larger than 0)

Ray Class

```
class Ray {
public:
        glm::vec3 Origin;
        glm::vec3 Direction;
};
```

NOTE: All data is public because the ray is a low-level class storing simple data.

Ray Intersection

- The fundamental operation in a ray tracer is the *ray intersection* routine
- The ray starts at a point known as the ray *origin* and shoots off in a 3D *direction*
- The ray shoots into a 3D environment with potentially millions of triangles or other primitives
- We are interested in knowing the *first* surface the ray hits
- The *ray intersection* routine takes a ray as input, as well as a scene containing geometry, and determines if and where the ray hits

```
bool Scene::Intersect(const Ray &ray,Intersection &hit);
```

Intersection Class

```
class Intersection {
public:
    Intersection()          {Mtl=0; Obj=0; HitDistance=1e10;}

    // Results of intersection test
    float HitDistance;
    glm::vec3 Position;
    glm::vec3 Normal;
    Material *Mtl;

    // Results of shading
    Color Shade;
};
```

NOTE: We will talk about Materials and Colors in a future lecture

Shading

- Once we have found an intersection (and the associated normal and material properties) we can do the process of *shading*
- Shading involves all of the calculations that compute the color reflected back along the initial ray towards the viewer
- This may (and often does) involve recursively tracing additional rays to determine shadows, reflections, and refractions

Ray Intersection & Shading

- Together, ray intersection and shading make up the two main routines in a ray tracer
- They are also the fundamental routines used in extensions such as path tracing and photon mapping

Ray Intersection Performance

- The fundamental issue of ray intersection is handling large scene complexity with reasonable performance
- Scenes often contain millions of triangles and one must trace millions of rays to render an image
- Consider a 800 x 600 image with two light sources and no reflective or refractive materials
- One must trace 800 x 600 = 480000 initial rays from the camera
- Assuming 2/3 of these hit objects (1/3 hit the sky), we then have to spawn two additional shadow rays, for an additional 640000 ray intersections and a total of 1.12 million rays
- High quality lighting and rendering effects often involve testing hundreds or thousands of rays *per pixel*
- Therefore, the performance of the ray intersection test is *critical*

Shading Performance

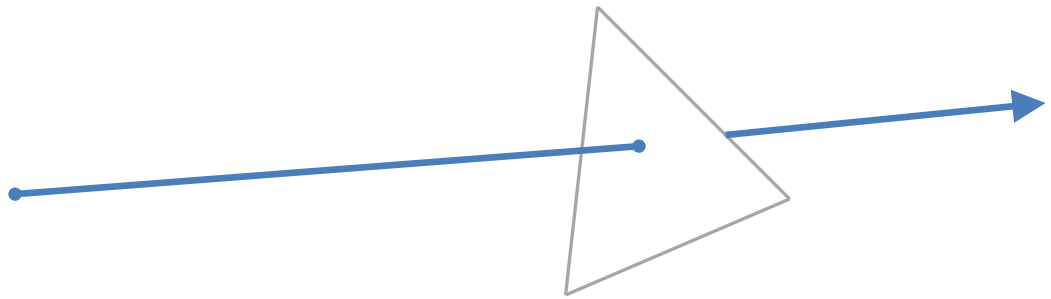
- Shading functions are called many times, but less than ray intersection (maybe 10%-50% as often)
- Shading functions often contain relatively quick computations and so aren't usually too expensive in themselves
- However, most shading involves recursively spawning more rays, which means more intersection and more shading
- Therefore, optimizing shading functions typically involves making them spawn as few rays as possible to produce the desired image quality

Triangles

- Triangles are used throughout computer graphics as the primitive of choice for rendering
- Most ray tracers *only* support triangles at the low level, and only perform actual ray intersection testing with triangles
- However, other surface types (such as curved surfaces, NURBS, subdivision surfaces, spheres, cylinders, implicit surfaces…) can still be supported as long as they can be *tessellated* into a bunch of small triangles
- This way, any surface type can be supported as long as they provide a tessellation routine
- Optionally, one could also implement ray intersection routines for each of those surface types, but in most cases, it significantly impacts the performance of ray intersection testing, and this is generally not done in commercial renderers

Ray-Triangle Intersection

- The ray-triangle intersection test is at the heart of the ray intersection problem, and often makes up the single biggest performance bottleneck in most ray tracers
- We will look at some ray-triangle intersection algorithms in an upcoming lecture



Vertex Class

```
class Vertex {
public:
        glm::vec3 Position;
        glm::vec3 Normal;
        glm::vec2 TexCoord;
};
```

Triangle Class

```
class Triangle {
public:
        Triangle();

        bool Intersect(const Ray &ray, Intersection &hit);

private:
        Vertex *Vtx[3];
        Material *Mtl;
};
```

Cameras

Camera Data

- The camera contains all of the information about how we are going to generate a 2D image from our 3D scene
- At a minimum, this includes:
  - Matrix: position & orientation of the camera
  - Field of view (FOV): determines the ‘zoom’ of the lens (i.e., ranges from wide angle to telephoto)
  - Resolution: x & y resolution of the image we want to render

Additional Camera Data

- Later in the quarter, we will extend the concept of a camera to include:
  - Focus range (depth of field)
  - Sub-pixel sampling (antialiasing)
  - Exposure settings (high dynamic range imaging)
  - Lens diffusion and imperfections
  - Shutter speed (motion blur)

Camera Matrix

- The camera matrix is a 3D matrix that positions the camera in the world
- Like any 3D matrix, it contains the **a**, **b**, **c**, and **d** column vectors
- The **d** vector is the position of the camera
- The **c** vector points backwards (-**c** is the direction of viewing)
- The **a** vector points to the right of the camera
- The **b** vector points to the up direction, relative to the camera
- The camera matrix is almost always orthonormal, so **a**, **b**, and **c** are unit length and perpendicular to each other

‘Look-At’ Function

- It is often convenient to define the camera matrix using a ‘look-at’ function
- This takes a camera **position**, a **target** object position, and a world ‘**up**’ vector (typically the y-axis), and then builds a matrix:

$$\text{glm} :: \text{vec3}\ \mathbf{d} = \mathbf{position}$$
$$\text{glm} :: \text{vec3}\ \mathbf{c} = \text{glm} :: \text{normalize}(\mathbf{d} - \mathbf{target})$$
$$\text{glm} :: \text{vec3}\ \mathbf{a} = \text{glm} :: \text{normalize}(\text{glm} :: \text{cross}(\mathbf{up}, \mathbf{c}))$$
$$\text{glm} :: \text{vec3}\ \mathbf{b} = \text{glm} :: \text{cross}(\mathbf{c}, \mathbf{a})$$

glm::mat4x4 cam(a.x,a.y,a.z,0, b.x,b.y,b.z,0, c.x,c.y,c.z,0, d.x,d.y,d.z,1)

- Note: **b** will be length 1.0 automatically, being the cross product of two orthogonal unit vectors
- Also note that glm has a lookAt function, but theirs computes the ‘view’ matrix, which is the inverse of the ‘camera’ matrix

Field of View

- If we are rendering a rectangular image, then the viewable camera volume is shaped like a pyramid, with the tip at the camera position
- We therefore have two relevant *field of view (FOV)* angles- one for the horizontal FOV and one for the vertical FOV
- The image itself is a rectangle, and therefore, its shape can be defined by as *aspect ratio*, which is the ratio of the image width to the height:

$$aspect = \frac{width}{height}$$

- However, it is important to notice that this is not the same as the ratio of the horizontal to vertical FOVs:

$$aspect \neq \frac{hfov}{vfov}$$

- Because the final image aspect ratio is typically determined by the viewing format (HDTV, film, etc.), it is useful to be able to set up the virtual camera lens by specifying the aspect and one of the FOVs

- It is traditional to define a camera lens by its vertical FOV and image aspect ratio

Field of View

- By examining some right triangles, the relationship between vertical and horizontal field of view can be found:

$$hfov = 2 \cdot \tan^{-1}\left(aspect \cdot \tan\frac{vfov}{2}\right)$$

Pixel Aspect

- The image aspect ratio is the ratio of the width to the height of the actual image
- However, this says nothing about the aspect ratio of the actual pixels of the image
- On most modern display hardware (monitors, tablets, phones…) the pixels themselves are square (aspect 1.0)
- If the pixels are square then the image aspect ratio is the same as the ratio of the horizontal to vertical resolution XRes/YRes
- However, it isn't too uncommon to come across display systems with non-square pixels
- In that case, the actual pixel aspect ratio is:

$$\text{Pixel Aspect} = (\text{ImageWidth} * \text{YRes}) / (\text{ImageHeight} * \text{XRes})$$

Camera Class

```
class Camera {
public:
        Camera();

        void SetFOV(float f);
        void SetAspect(float a);
        void SetResolution(int x,int y);
        void LookAt(glm::vec3 &pos,glm::vec3 &target,glm::vec3 &up);

        void Render(Scene &s);
        void SaveBitmap(char *filename);
private:
        int XRes,YRes;
        glm::mat4x4 WorldMatrix;
        float VerticalFOV;
        float Aspect;
        Bitmap BMP;
};
```

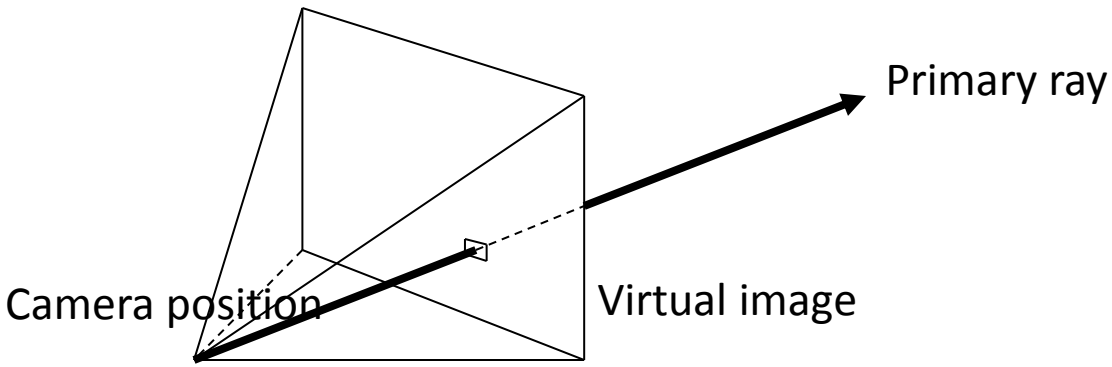
Image Rendering

- When we ray-trace an image, we start by generating rays at the camera and shoot them through each pixel into the scene
- For example, we have a loop something like this:

```
Camera::Render() {
        int x,y;
        for(y=0; y<YRes; y++) {
                for(x=0; x<XRes; x++) {
                        RenderPixel(x,y);
                }
        }
}
```

Camera Rays

- We start by ‘shooting’ rays from the camera out into the scene
- We can render the pixels in any order we choose (even in random order!), but we will keep it simple and go from top to bottom, and left to right
- We loop over all of the pixels and generate an initial *primary ray* (also called a *camera ray* or *eye ray*)
- The ray origin is simply the camera’s position in world space
- The direction is computed by first finding location of a ‘virtual pixel’ on a ‘virtual image plane’, and then computing a normalized direction from the camera position to the virtual pixel



Project 1

- Project 1 will be a basic ray tracer that can render a box
- It will be defined in detail in the next lecture
- If you want to get started early, you can start with the Ray, Intersection, Vertex, and Triangle classes defined within and implement the function Triangle::Intersect()